Coursework 4

This coursework is worth 15% and is due on 2nd January at 16:00. You are asked to implement a compiler for the WHILE language that targets the assembler language provided by Jasmin or Krakatau (both have a very similar syntax). You can do the implementation in any programming language you like, but you need to submit the source code with which you answered the questions, otherwise a mark of 0% will be awarded. You should use the lexer and parser from the previous courseworks. Implement your compiler in the file `cw04.sc`.

⚠ Disclaimer

It should be understood that the work you submit represents your own effort. You have not copied from anyone else. An exception is the Scala code I showed during the lectures, which you can use. You can also use your own code from the CW 1, CW 2 and CW 3. But do not be tempted to ask Github Copilot for help or do any other shenanigans like this!

Jasmin Assembler

For this coursework you will need an assembler. The Jasmin assembler is available from

<div align="center">

http://jasmin.sourceforge.net

</div>

This is a jar-file you can run on the commandline. There is a user guide for Jasmin

<div align="center">

http://jasmin.sourceforge.net/guide.html

</div>

and also a description of some of the instructions that the JVM understands

<div align="center">

http://jasmin.sourceforge.net/instructions.html

</div>

If you generated a correct assembler file for Jasmin, for example `loops.j`, you can use

<div align="center">

```
java -jar jasmin.jar loops.j
```

</div>

in order to translate it into Java Byte Code. If needed, you need to give the path to the Jasmin jar-file. The resulting class file can be run with

<div align="center">

```
java loops
```

</div>

where you might need to give the correct path to the class file. For example:

<div align="center">

```
java -cp . loops/loops
```

</div>

There are also other resources about Jasmin on the Internet, for example

https:
//saksagan.ceng.metu.edu.tr/courses/ceng444/link/f3jasmintutorial.html

and

http://www.csc.villanova.edu/~tway/courses/csc4181/s2022/labs/
finalproject/JVM.pdf

If possible use Jasmin for the coursework. The Krakatau assembler below has a slightly different syntax.

Krakatau Assembler (Version 1 & 2)

The Krakatau assembler is available from

https://github.com/Storyyeller/Krakatau/tree/master

This assembler requires Python and a package called `ply` available from

https://pypi.python.org/pypi/ply

This assembler is largely compatible with the Jasmin syntax—that means for the files we are concerned with here, it understands the same input syntax (no changes to your compiler need to be made; ok maybe some small syntactic adjustments are needed, for example labels need to start with a capital 'L'). You can generate Java Byte Code by using

```
python Krakatau-master/assemble.py loops.j
```

where you may have to adapt the directory where Krakatau is installed (I just downloaded the zip file from Github and `Krakatau-master` was the directory where it was installed). Again the resulting class-file you can run with `java`. There is also a newer version of Krakatau available at

https://github.com/Storyyeller/Krakatau/tree/v2

This is now a Rust program using Cargo as package manager (I have not tried this version—I assume it should produce the same output, but might be easier to install because it avoids Python's *dependency hell*).

Question 1

You need to lex and parse WHILE programs, and then generate Java Byte Code instructions for the Jasmin assembler (or Krakatau assembler). For this you should use the ASTs defined in CW3 (including logical operators). As part of the solution you need to submit the assembler instructions for the Fibonacci and Factorial programs.

```
write "Fib: ";
read n;
minus1 := 1;
minus2 := 0;
while n > 0 do {
        temp := minus2;
        minus2 := minus1 + minus2;
        minus1 := temp;
        n := n - 1
};
write "Result: ";
write minus2 ;
write "\n"
```

Figure 1: The Fibonacci program in the WHILE language.

Question 2

Extend the syntax of your language so that it contains also `for`-loops, like

$$\texttt{for } Id := AExp \texttt{ upto } AExp \texttt{ do } Block$$

The intended meaning is to first assign the variable *Id* the value of the first arithmetic expression, then test whether this value is less or equal than the value of the second arithmetic expression. If yes, go through the loop, and at the end increase the value of the loop variable by 1 and start again with the test. If no, leave the loop. For example the following instance of a `for`-loop is supposed to print out the numbers 2, 3, 4.

```
for i := 2 upto 4 do {
    write i
}
```

There are two ways how this can be implemented: one is to adapt the code generation part of the compiler and generate specific code for `for`-loops; the other is to translate the abstract syntax tree of `for`-loops into an abstract syntax tree using existing language constructs. For example the loop above could be translated to the following `while`-loop:

```
i := 2;
while (i <= 4) do {
    write i;
    i := i + 1
}
```

Question 3 (OPTIONAL)

In this question you are asked to think about the following program:

```
for i := 1 upto 10 do {
  for i := 1 upto 10 do {
    write i
  }
}
```

Note that in this program the variable i is used twice. Therefore you need to make a decision about how it should be compiled? What should the program print?

Question 4

Extend the lexer and parser to add a break keyword. Modify the compiler (including lexer and parser) such that when a break-statement is encountered the code should jump out of the "enclosing" for/while-loop, or in case it is not inside such a loop to the end of the program. For example the program

```
// should print 0 .. 10
for i := 0 upto 10 do {
  write i;
  write "\n"
};

// should print 0 .. 4
for i := 0 upto 10 do {
  if i > 4 then break else skip;
  write i;
  write "\n"
};

write "Should print\n";
break;
write "Should not print\n"
```

should print out 0 to 10 with the first for-loop, but only 0 to 4 in the second. Similarly it should print out "Should print", but not "Should not print". For this you need to add a label to the end of every for- and while-loop and also to the end of the whole program just in case you need to jump to that label via a break. The file you need to be able to process for this question is called break.while.

Further Information

The Java infrastructure unfortunately does not contain an assembler out-of-the-box (therefore you need to download the additional package Jasmin or Krakatau—see above). But it does contain a disassembler, called `javap`. A disassembler does the "opposite" of an assembler: it generates readable assembler code from Java Byte Code. Have a look at the following example: Compile using the usual Java compiler the simple Hello World program below:

```
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

You can use the command

```
javap -c HelloWorld
```

to see the assembler instructions of the Java Byte Code that has been generated for this program. You can compare this with the code generated for the Scala version of Hello World.

```
object HelloWorld {
  def main(args: Array[String]) = {
     println("Hello World!")
  }
}
```

Library Functions

You need to generate code for the commands `write` and `read`. This will require the addition of some "library" functions to your generated code. The first command even needs two versions, because you need to write out an integer and string. The Java byte code will need two separate functions for this. For writing out an integer, you can use the assembler code

```
.method public static write(I)V
  .limit locals 1
  .limit stack 2
  getstatic java/lang/System/out Ljava/io/PrintStream;
  iload 0
  invokevirtual java/io/PrintStream/println(I)V
  return
.end method
```

This function will invoke Java's `println` function for integers. Then if you need to generate code for `write x` where x is an integer variable, you can generate

```
iload n
invokestatic XXX/XXX/write(I)V
```

where `n` is the index where the value of the variable x is stored. The `XXX/XXX` needs to be replaced with the class name which you use to generate the code (for example `fib/fib` in case of the Fibonacci numbers).

Writing out a string is similar. The corresponding library function uses strings instead of integers:

```
.method public static writes(Ljava/lang/String;)V
    .limit stack 2
    .limit locals 1
    getstatic java/lang/System/out Ljava/io/PrintStream;
    aload 0
    invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
    return
.end method
```

The code that needs to be generated for `write "some_string"` commands is

```
ldc "some_string"
invokestatic XXX/XXX/writes(Ljava/lang/String;)V
```

Again you need to adjust the `XXX/XXX` part in each call.

The code for `read` is more complicated. The reason is that inputting a string will need to be transformed into an integer. The code in Figure 2 does this. It can be called with

```
invokestatic XXX/XXX/read()I
istore n
```

where `n` is the index of the variable that requires an input.

```
1   .method public static read()I
2       .limit locals 10
3       .limit stack 10
4
5       ldc 0
6       istore 1  ; this will hold our final integer
7   Label1:
8       getstatic java/lang/System/in Ljava/io/InputStream;
9       invokevirtual java/io/InputStream/read()I
10      istore 2
11      iload 2
12      ldc 10  ; test for the newline delimiter for Unix
13      isub
14      ifeq Label2
15      iload 2
16      ldc 13  ; test for the carriage-return in Windows
17      isub
18      ifeq Label2
19      iload 2
20      ldc 32   ; the space delimiter
21      isub
22      ifeq Label2
23      iload 2
24      ldc 48   ; we have our digit in ASCII, have to subtract it from 48
25      isub
26      ldc 10
27      iload 1
28      imul
29      iadd
30      istore 1
31      goto Label1
32  Label2:
33      ; when we come here we have our integer computed
34      ; in local variable 1
35      iload 1
36      ireturn
37  .end method
```

Figure 2: Assembler code for reading an integer from the console. This code is portable for Unix and Windows (see Lines 11–18 for 2 separate tests for the various end-of-line markers). Thanks to Harry Dilnot to make it portable.